

15-213 Attack Lab

June 13, 2018

Agenda

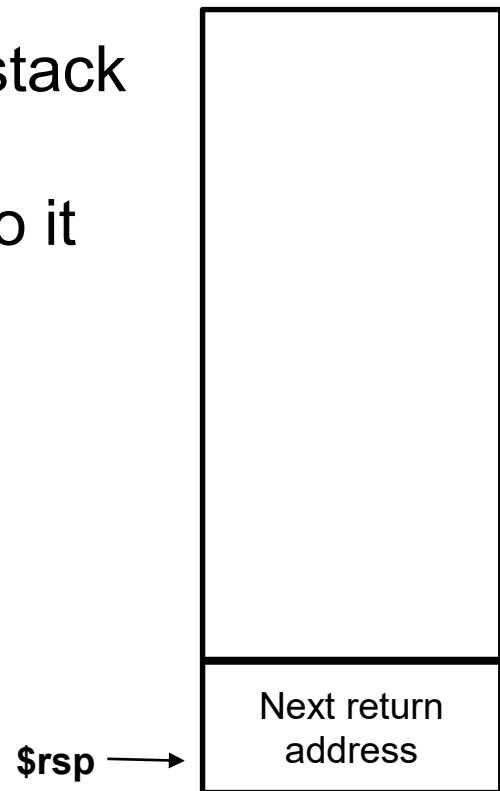
- **Buffer Overflow Attacks**
- **Attack Lab Activities**

Attack Lab

- We're letting you hijack programs by running buffer overflow attacks on them...
- To understand stack discipline and stack frames
- To defeat relatively secure programs with return oriented programming

Stack Smashing Attack

- *Callq* pushes the return address onto the stack
- *Retq* pops this return address and jumps to it



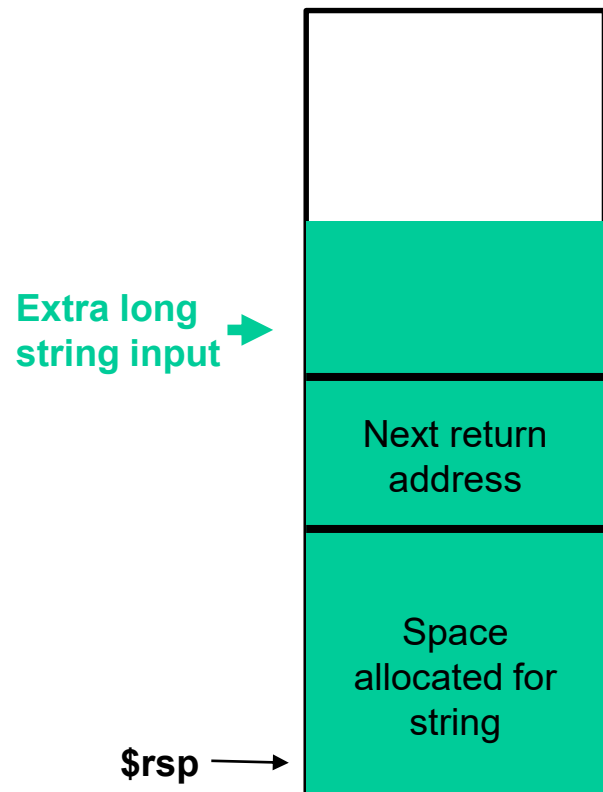
Buffer Overflows

- Local string variables are stored on the stack
- C functions do not do size checking of strings



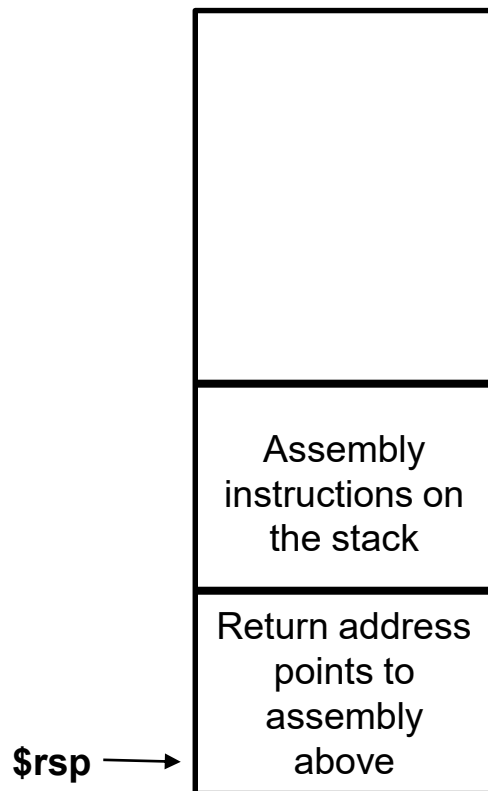
Buffer Overflows

- You can write a string that overwrites the return address
- Activity 1 steps through an example of overwriting the return address on the stack



Executing Commands on the Stack

- What if instead of jumping to a predefined function, we jumped to code on the stack?
- Activity 2 steps through an example of executing code on the stack



OS Countermeasures

- Executable code is not allowed on the stack (unless we specifically allow it – e.g. through `mprotect` like we do for activity 2)
- Thus, we have to use executable code already in the program to do what we want
- But code often doesn't already contain our exploit function – so what can we do instead?

Return-Oriented Programming

- Goal: execute a small section of code, return, call another small section of code. Repeat until you execute your exploit
- Activity 3 steps you through an example of a return oriented programming exploit

Attack Lab Activities

■ Three activities

- Each relies on a specially crafted assembly sequence to purposefully overwrite the stack

■ Activity 1 – Overwrites the return addresses (Buffer Overflow)

■ Activity 2 – Writes assembly instructions onto the stack

■ Activity 3 – Uses byte sequences in libc as the instructions (Return-Oriented Programming)

Attack Lab Activities

- Work in pairs: one student needs a laptop
- Login to a shark machine

```
$ wget http://www.cs.cmu.edu/~213/activities/lec11a.tar
```

```
$ tar xf lec11a.tar
```

```
$ cd lec11a
```

```
$ make
```

```
$ gdb act1
```

Activity 1

(gdb) break clobber

(gdb) run

(gdb) x \$rsp

(gdb) backtrace

Q. Does the value at the top of the stack match any frame?

A. 0x400553 is the address to return to in main

Activity 1 Continued

```
(gdb) x /2gx $rdi // Here are the two key values
```

```
(gdb) stepi // Keep doing this until
```

```
(gdb)
clobber () at support.s:16
16          ret
```

```
(gdb) x $rsp
```

Q. Has the return address changed?

A. 0x400500 was the first number pointed to by \$rdi

```
(gdb) finish // Should exit and print out “Hi!”
```

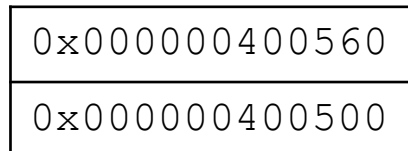
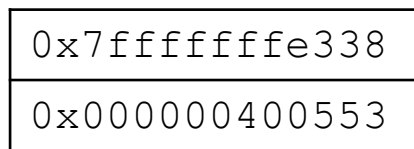
Activity 1 Post

- Clobber overwrites part of the stack with memory at `$rdi`, including the all-important return address
- In `act1`, it writes two new return addresses:
 - `0x400500`: address of `printHi()`
 - `0x400560`: address in `main`

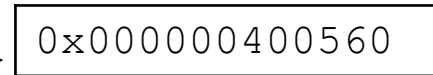
Call `clobber()`



Clobber executes



ret



ret

In `printHi()`

In `main()`

Activity 2

```
$ gdb act2
```

```
(gdb) break clobber
```

```
(gdb) run
```

```
(gdb) x $rsp
```

Q. What is the address of the stack and the return address?

A. 0x7fffffffef018 -> 0x40058a

```
(gdb) x /4gx $rdi
```

Q. What will the new return address be?

A. 0x7fffffffef020 (First address stored using \$rdi)

Activity 2 Continued

(gdb) x /5i \$rdi + 8 // Display as instructions

Q. Why \$rdi + 8?

A. Want to ignore the 8-byte return address

Q. What are the three addresses?

A. 0x48644d, 0x4022e0, 0x4011a0

(gdb) break puts

(gdb) break exit

Q. Do these addresses look familiar?

A. puts – 0x4022e0, exit – 0x4011a0

Activity 2 Post

- **Normally programs cannot execute instructions on the stack**
 - Main used `mprotect` to disable the memory protection for this activity
- **Clobber wrote an address that's on the stack as a return address**
 - Followed by a sequence of instructions
 - Three addresses show up in the exploit:
 - `0x48644d` → “Hi\n” string
 - `0x4022e0` → `puts()` function
 - `0x4011a0` → `exit()` function

Activity 3

```
$ gdb act3
```

```
(gdb) break clobber
```

```
(gdb) run
```

```
(gdb) x /5gx $rdi
```

Q. Which value will be first on the stack? Why is this important?

A. 0x457d0c, this is the address to return to from clobber

Activity 3 Continued

(gdb) x /2i <return address>

Q. What does this sequence do?

A. Pops next stack value into \$rdi, then returns

Q. Check the other addresses. Note that some are return addresses and some are for data. When you continue, what will the code now do?

A. Print "Hi\n"

Activity 3 Post

- It's harder to stop programs from running existing pieces of code in the executable.
- Clobber wrote multiple return addresses (aka gadgets) that each performed a small task, along with data that will get popped off the stack while running the gadgets.
 - 0x457d0c: pop %rdi; retq
 - 0x47fa64: Pointer to the string "Hi\n"
 - 0x429a6a: pop %rax; retq
 - 0x400500: Address of a printing function
 - 0x47f001: callq *%rax

Activity 3 Post

- Note that some of the return addresses actually cut off bytes from existing instructions

```

457cfa: 48 83 c4 28      add    $0x28,%rsp
457cfe: 5b              pop    %rbx
457cff: 4a 8d 44 3d 00   lea   0x0(%rbp,%r15,1),%rax
457d04: 5d              pop    %rbp
457d05: 41 5c           pop    %r12
457d07: 41 5d           pop    %r13
457d09: 41 5e           pop    %r14
457d0b: 41 5f           pop    %r15
457d0d: c3             retq
457d0e: 48 83 7c 24 10 00  cmpq  $0x0,0x10(%rsp)
457d14: 74 8a           je    457ca0 <_IO_getline_info+0xd0>

```

0x457d0b ...0c ...0d

pop %r15 retq

41 5f c3

pop %rdi retq

5f c3

Operation	Register <i>R</i>							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq <i>R</i>	58	59	5a	5b	5c	5d	5e	5f

If you get stuck...

- ***Please read the writeup!***
- CS:APP Chapter 3
- View lecture notes and course FAQ at <http://www.cs.cmu.edu/~213>
- Post a **private** question on Piazza
- `man gdb` – `gdb`'s help command

Attack Lab Tools

- **`gcc -c test.s; objdump -d test.o > test.asm`**

Compiles the assembly code in test.s and shows the actual bytes for the instructions

- **`./hex2raw < exploit.txt > converted.txt`**

Convert hex codes in exploit.txt into raw ASCII strings to pass to targets
See the writeup for more details on how to use this

- **`(gdb) display /12gx $rsp` `(gdb) display /2i $rip`**

Displays 12 elements on the stack and the next 2 instructions to run

GDB is also useful for tracing to see if an exploit is working